

10707

Deep Learning

Russ Salakhutdinov

Machine Learning Department
rsalakhu@cs.cmu.edu

<http://www.cs.cmu.edu/~rsalakhu/10707/>

Neural Networks I

Neural Networks Online Course

- **Disclaimer:** Much of the material and slides for this lecture were borrowed from Hugo Larochelle's class on Neural Networks:
<https://sites.google.com/site/deeplearningsummerschool2016/>

http://info.usherbrooke.ca/hlarochelle/neural_networks

- Hugo's class covers many other topics: convolutional networks, neural language model, Boltzmann machines, autoencoders, sparse coding, etc.

- We will use his material for some of the other lectures.

Click with the mouse or tablet to draw with pen 2

RESTRICTED BOLTZMANN MACHINE

Topics: RBM, visible layer, hidden layer, energy function

The diagram illustrates the architecture of a Restricted Boltzmann Machine (RBM). It consists of two layers of binary units: a hidden layer (h) and a visible layer (x). The hidden layer units are represented by circles with bias terms b_j inside. The visible layer units are represented by circles with bias terms c_k inside. Connections between the hidden and visible layers are represented by a weight matrix W . The diagram also shows the bias terms b_j and c_k and the weight matrix W .

Energy function:
$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{h}^\top \mathbf{W} \mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h}$$
$$= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j$$

Distribution: $p(\mathbf{x}, \mathbf{h}) = \exp(-E(\mathbf{x}, \mathbf{h})) / Z$ ← partition function (intractable)

A small video inset in the bottom right corner shows a person with glasses and a beard, wearing a red and blue shirt, speaking.

Feedforward Neural Networks

- ▶ How neural networks predict $f(\mathbf{x})$ given an input \mathbf{x} :

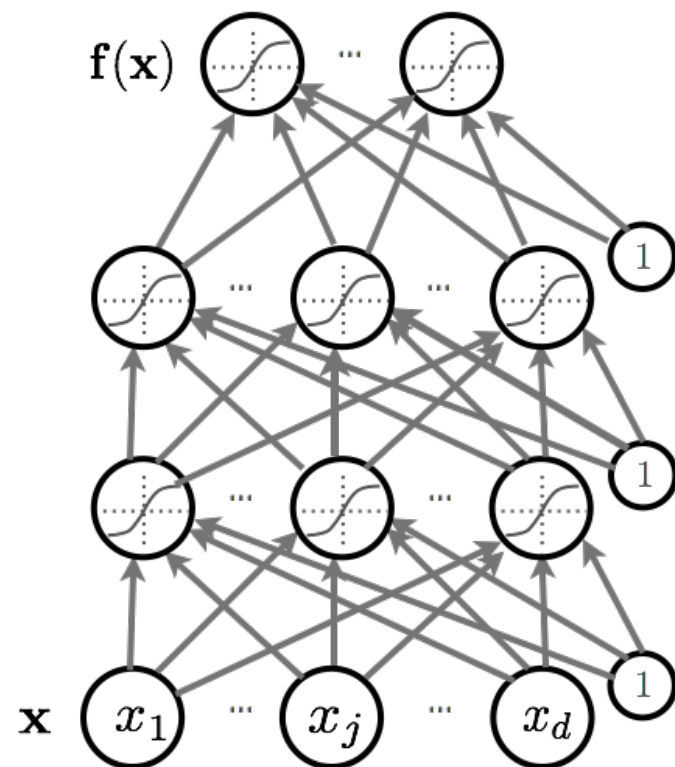
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



Feedforward Neural Networks

- ▶ How neural networks predict $f(\mathbf{x})$ given an input \mathbf{x} :

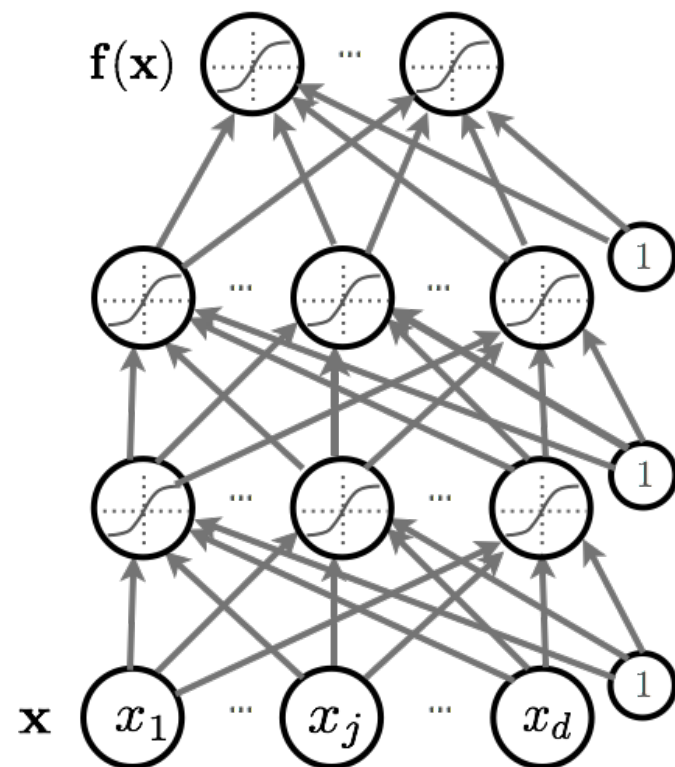
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



Artificial Neuron

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron output activation:

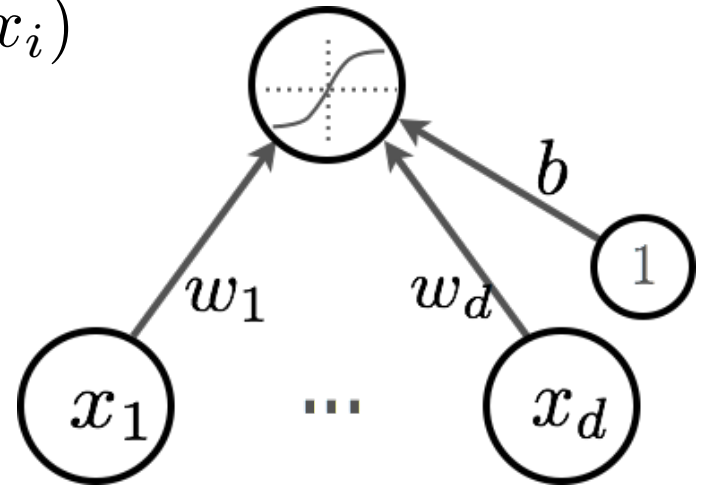
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

where

\mathbf{W} are the weights (parameters)

b is the bias term

$g(\cdot)$ is called the activation function

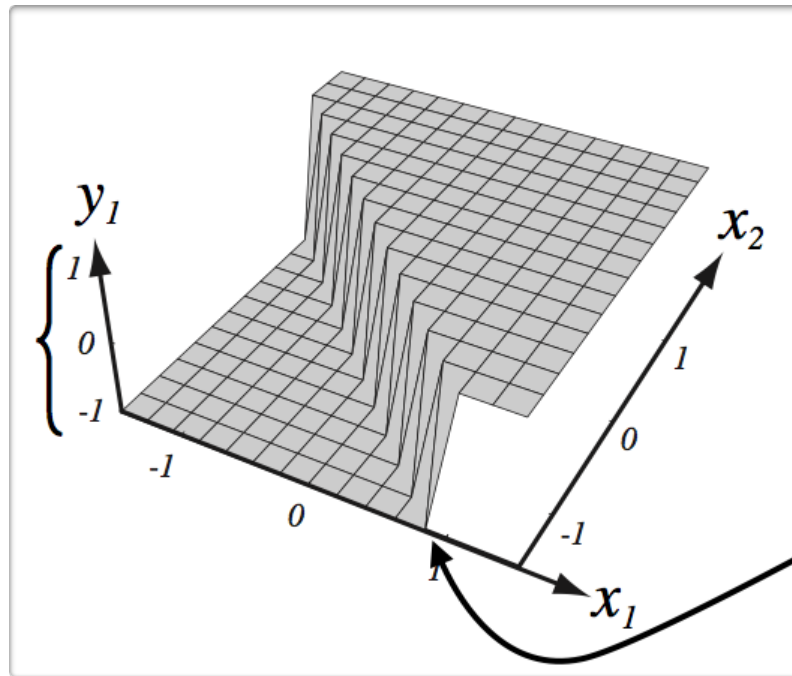


Artificial Neuron

- Output activation of the neuron:

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

Range is
determined
by $g(\cdot)$



Bias only changes
the position of the
ruff

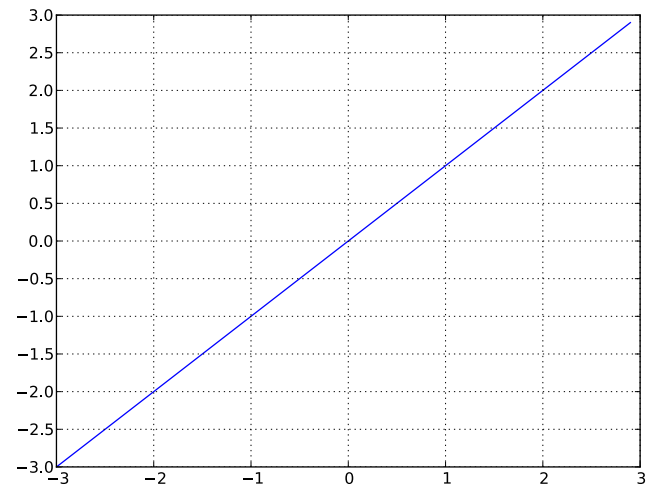
(from Pascal Vincent's slides)

Activation Function

- Linear activation function:

- No nonlinear transformation
- No input squashing

$$g(a) = a$$

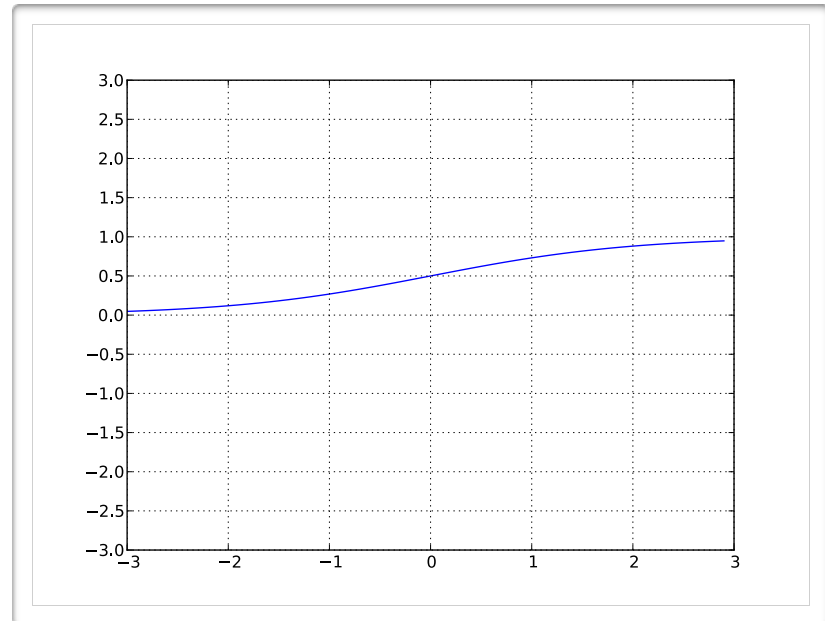


Activation Function

- Sigmoid activation function:

- Squashes the neuron's output between 0 and 1
- Always positive
- Bounded
- Strictly Increasing

$$g(a) = \text{sigm}(a) = \frac{1}{1 + \exp(-a)}$$



Activation Function

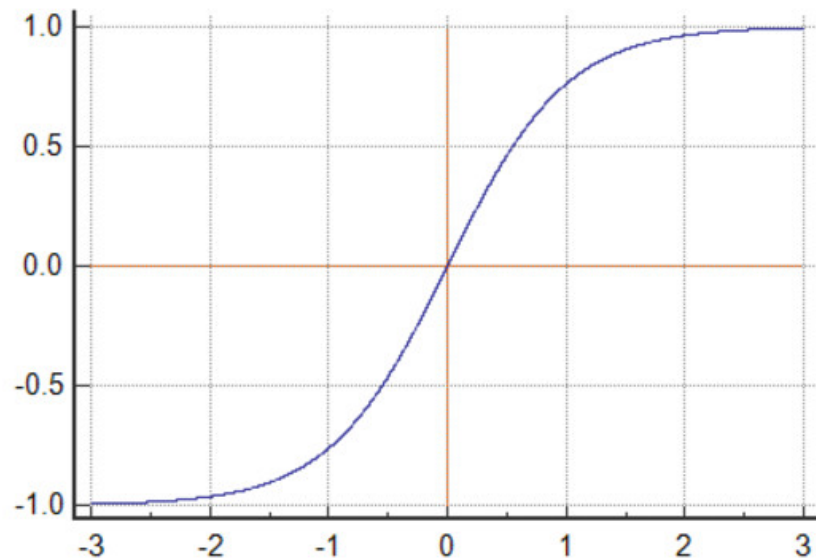
- Hyperbolic tangent (“tanh”) activation function:

- Squashes the neuron’s activation between -1 and 1

$$g(a) = \tanh(a) =$$

$$= \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

- Can be positive or negative
- Bounded
- Strictly increasing
(wrong plot)

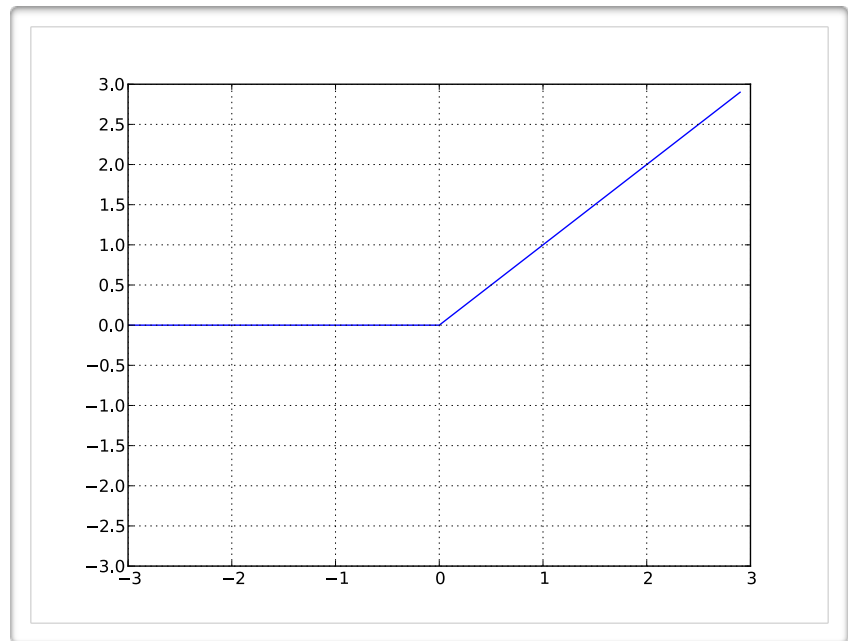


Activation Function

- Rectified linear (ReLU) activation function:

- Bounded below by 0 (always non-negative)
- Tends to produce units with sparse activities
- Not upper bounded
- Strictly increasing

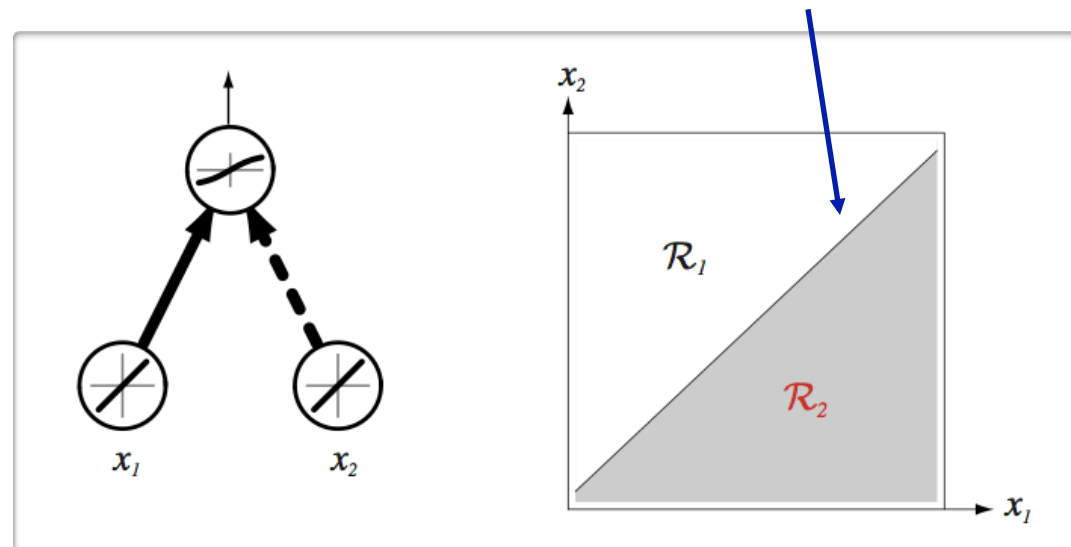
$$g(a) = \text{reclin}(a) = \max(0, a)$$



Decision Boundary of a Neuron

- Binary classification:
 - With sigmoid, one can interpret neuron as estimating $p(y = 1 | \mathbf{x})$
 - Interpret as a **logistic classifier**

- If activation is greater than 0.5, predict 1
- Otherwise predict 0

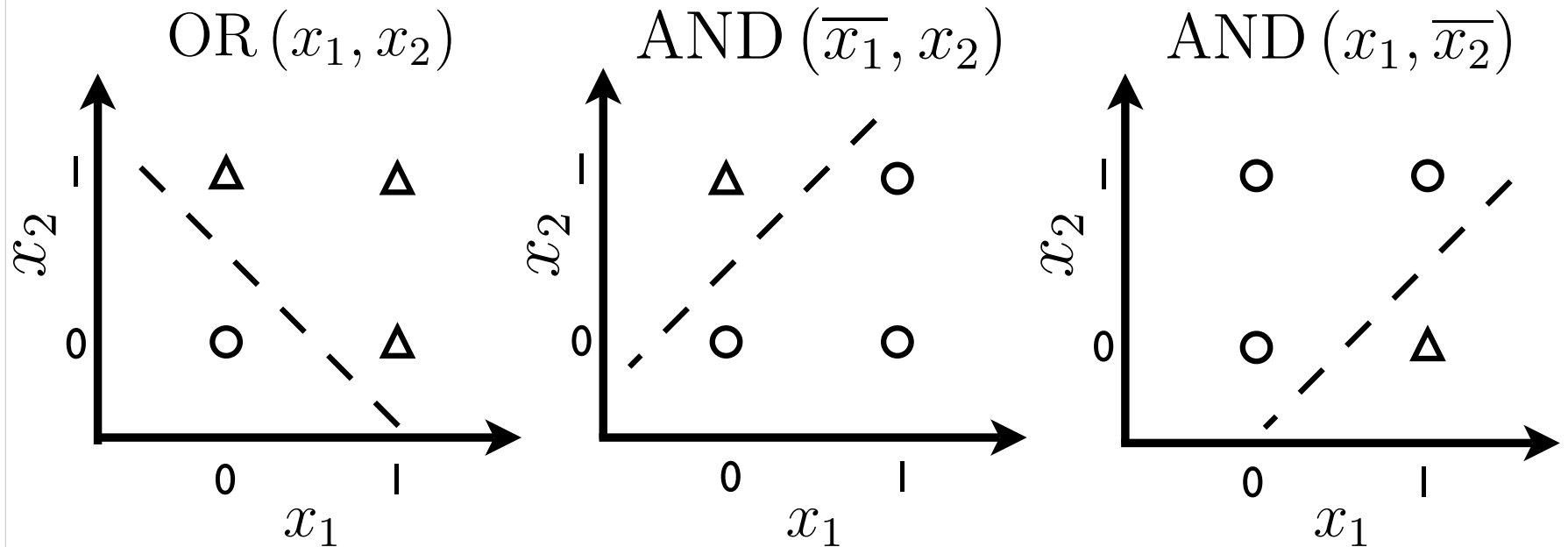


(from Pascal Vincent's slides)

Same idea can be applied
to a tanh activation

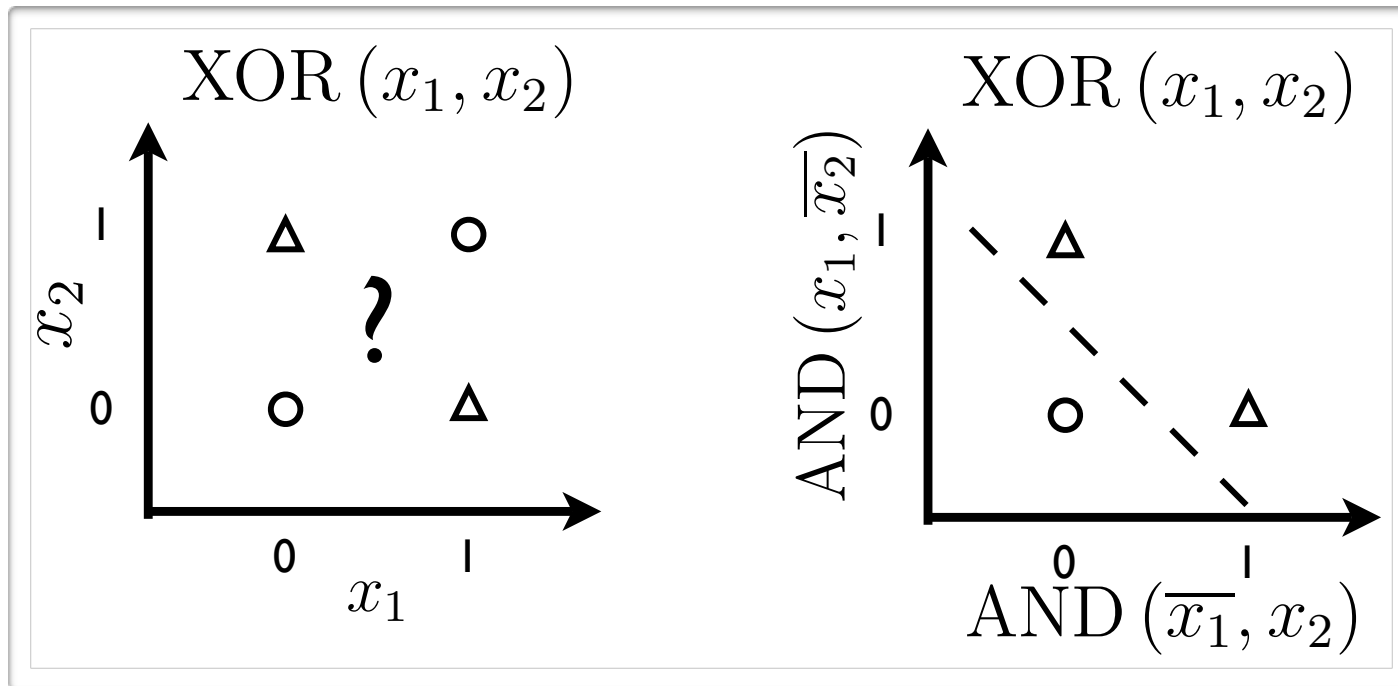
Capacity of a Single Neuron

- Can solve linearly separable problems.



Capacity of a Single Neuron

- Can not solve non-linearly separable problems.



- Need to transform the input into a better representation.
- Remember **basis functions**!

Single Hidden Layer Neural Net

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j)$$

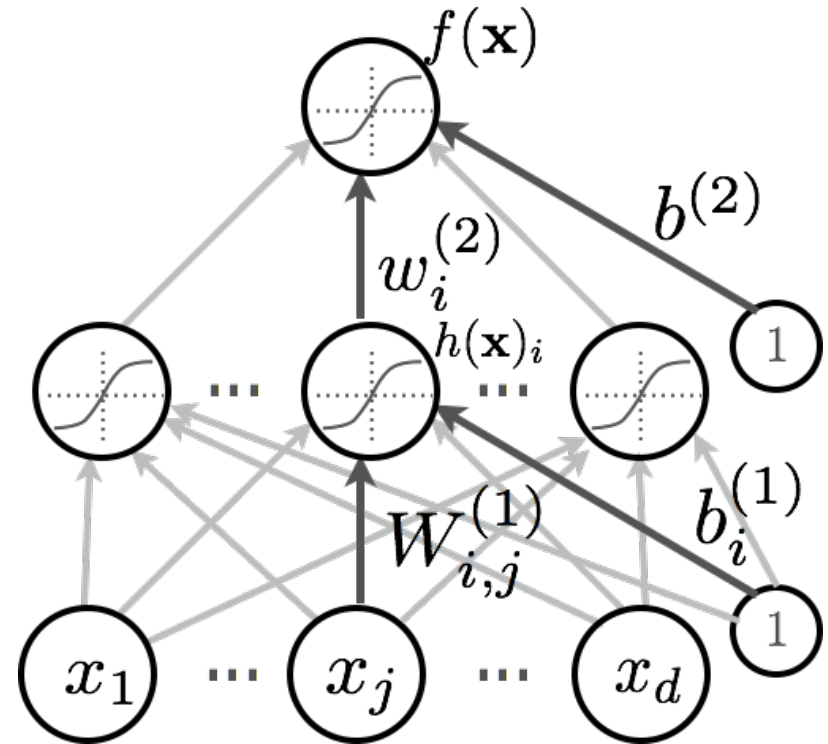
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o \left(b^{(2)} + \mathbf{w}^{(2)\top} \mathbf{h}^{(1)} \mathbf{x} \right)$$

Output activation
function



Softmax Activation Function

- ▶ Remember **multi-way classification**:
 - We need multiple outputs (1 output per class)
 - We need to estimate conditional probability: $p(y = c|\mathbf{x})$
 - Discriminative Learning

- ▶ Softmax activation function at the output

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$

- strictly positive
 - sums to one
- ▶ Predict class with the highest estimated class conditional probability.

Multilayer Neural Net

- Consider a network with L hidden layers.

- layer pre-activation for $k > 0$

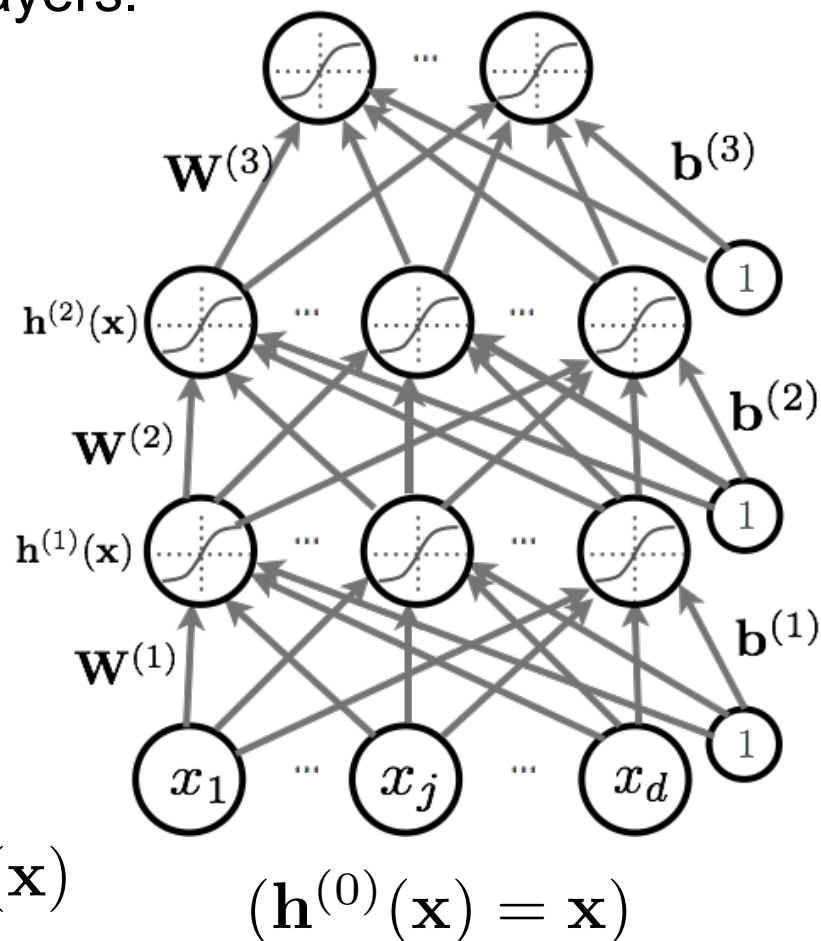
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation from 1 to L :

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

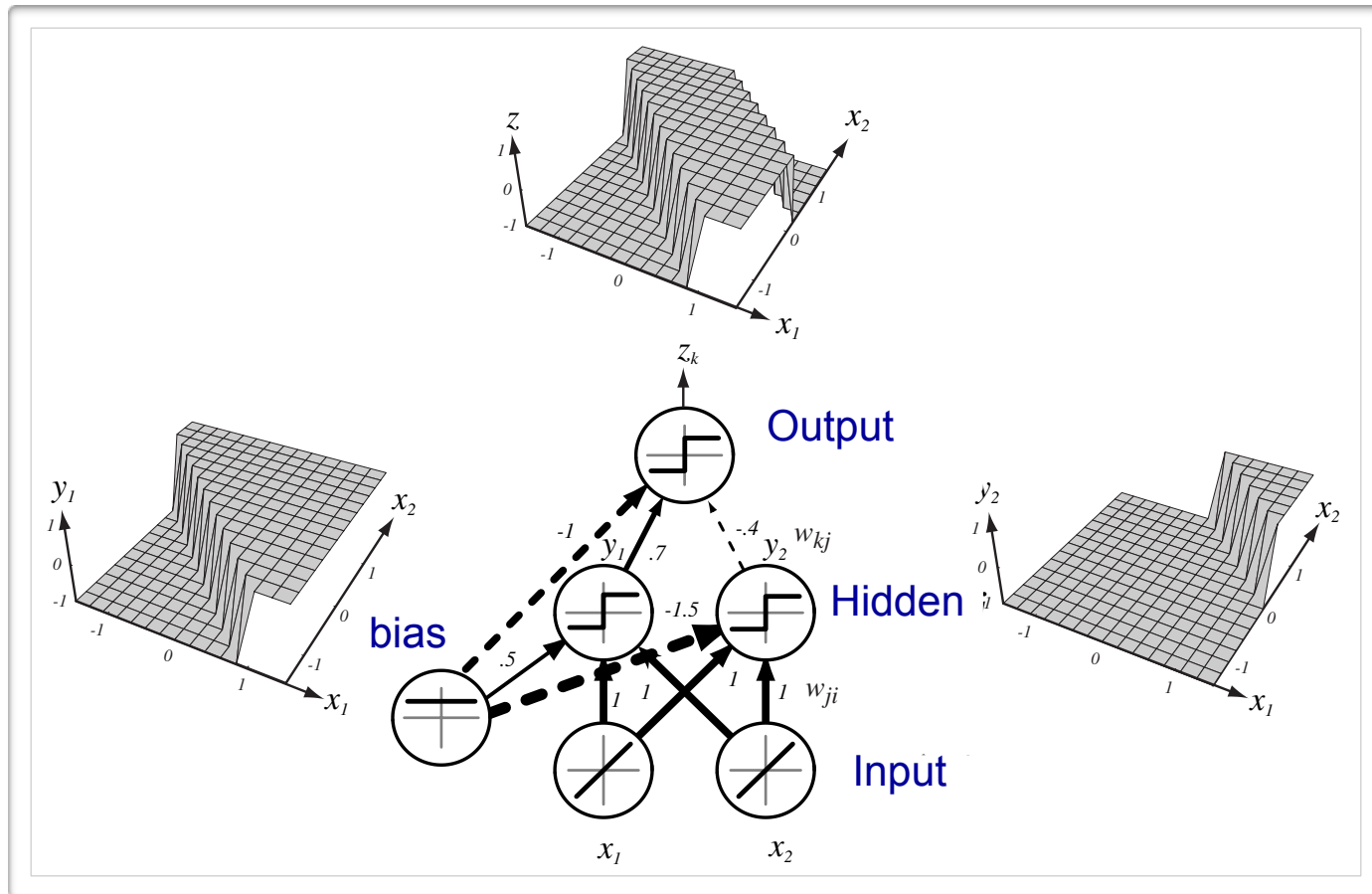
- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Capacity of Neural Nets

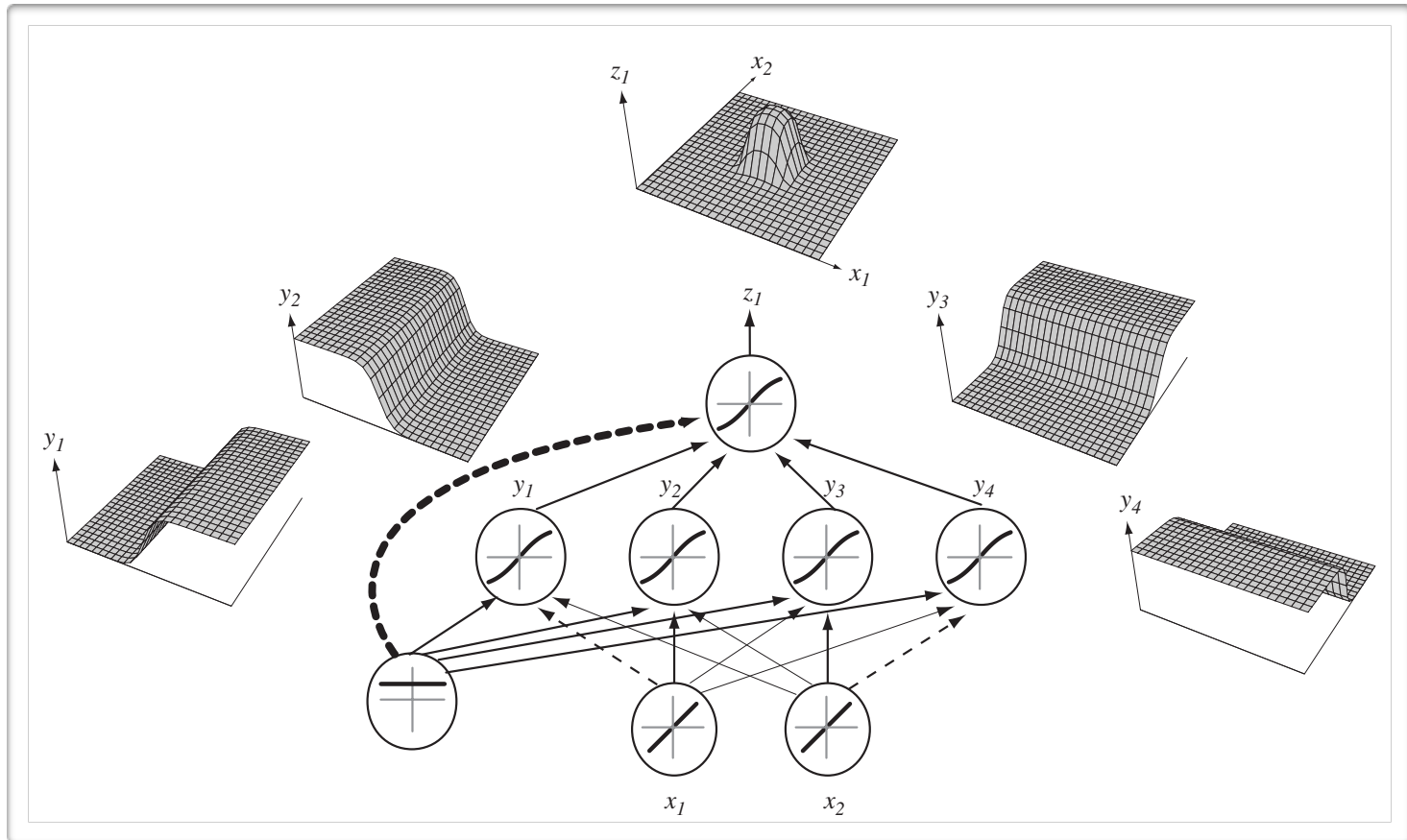
- Consider a single layer neural network



(from Pascal Vincent's slides)

Capacity of Neural Nets

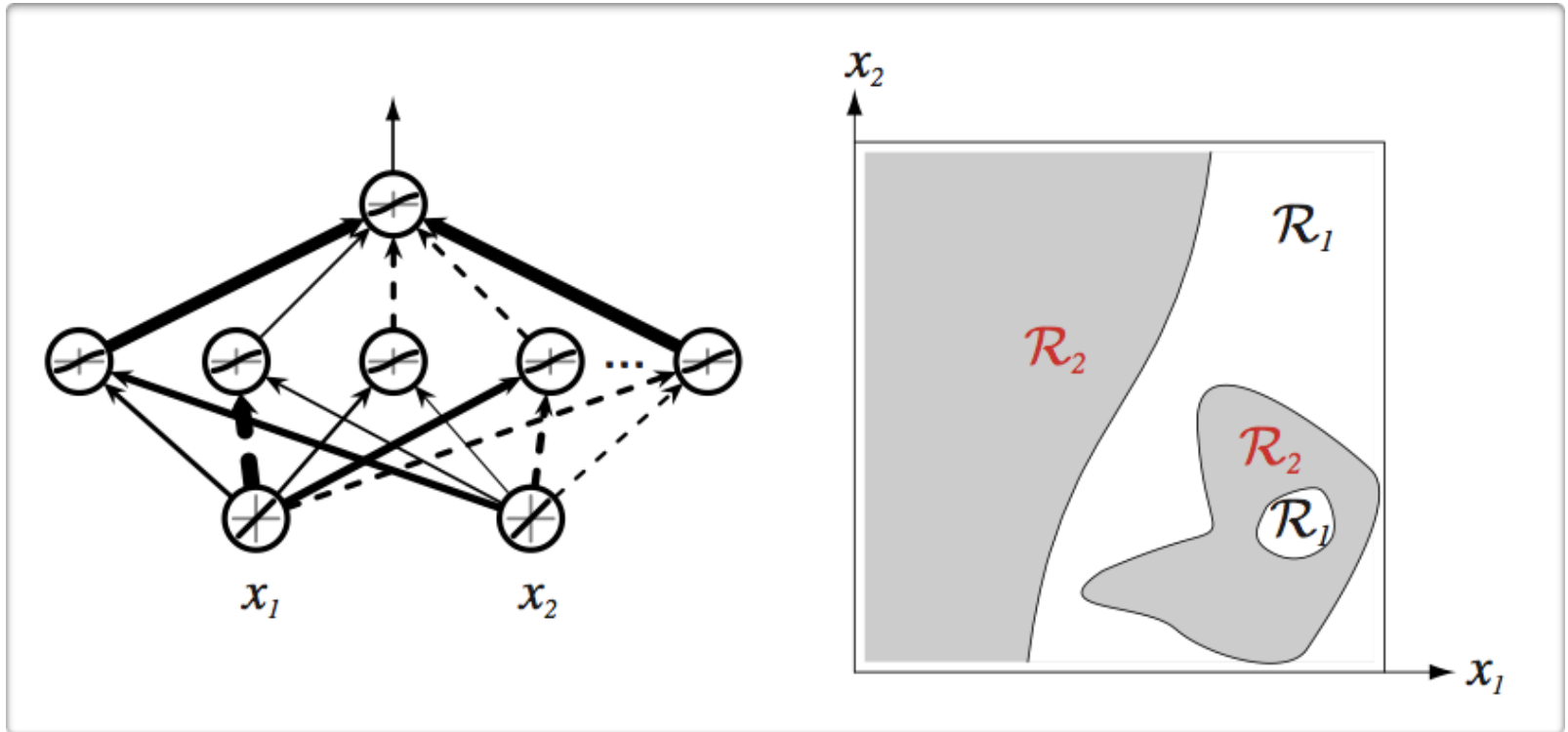
- Consider a single layer neural network



(from Pascal Vincent's slides)

Capacity of Neural Nets

- Consider a single layer neural network



(from Pascal Vincent's slides)

Universal Approximation

- Universal Approximation Theorem (Hornik, 1991):
 - “a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units”
- This applies for sigmoid, tanh and many other activation functions.
- However, this does not mean that there is learning algorithm that can find the necessary parameter values.

Feedforward Neural Networks

- ▶ How neural networks predict $f(\mathbf{x})$ given an input \mathbf{x} :

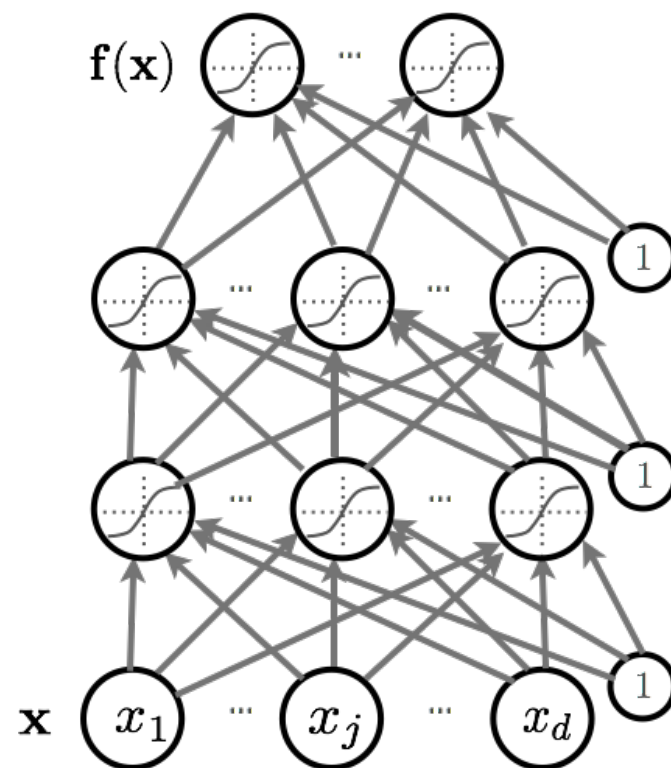
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



Training

- Empirical Risk Minimization:

$$\arg \min_{\boldsymbol{\theta}} \frac{1}{T} \sum_t \underbrace{l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})}_{\text{Loss function}} + \underbrace{\lambda \Omega(\boldsymbol{\theta})}_{\text{Regularizer}}$$

- Learning is cast as optimization.
 - For classification problems, we would like to minimize classification error.
 - Loss function can sometimes be viewed as **a surrogate for what we want to optimize** (e.g. upper bound)

Stochastic Gradient Descend

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For $t=1:T$

- for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:

- **Loss function:** $l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- A procedure to **compute gradients:** $\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- **Regularizer** and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

Loss Function

- Let us start by considering a classification problem with a softmax output layer.
- We need to estimate: $f(\mathbf{x})_c = p(y = c|\mathbf{x})$
 - We can maximize the log-probability of the correct class given an input: $\log p(y^{(t)} = c|x^{(t)})$
- Alternatively, we can minimize the negative log-likelihood:
$$l(\mathbf{f}(\mathbf{x}), y) = - \sum_c 1_{(y=c)} \log f(\mathbf{x})_c = - \log f(\mathbf{x})_y$$
- As seen before, this is also known as a **cross-entropy entropy function** for multi-class classification problem.

Stochastic Gradient Descend

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For $t=1:T$

- for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:

- Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- A procedure to compute gradients: $\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- Regularizer and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

Multilayer Neural Net: Reminder

- Consider a network with L hidden layers.

- layer pre-activation for $k > 0$

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

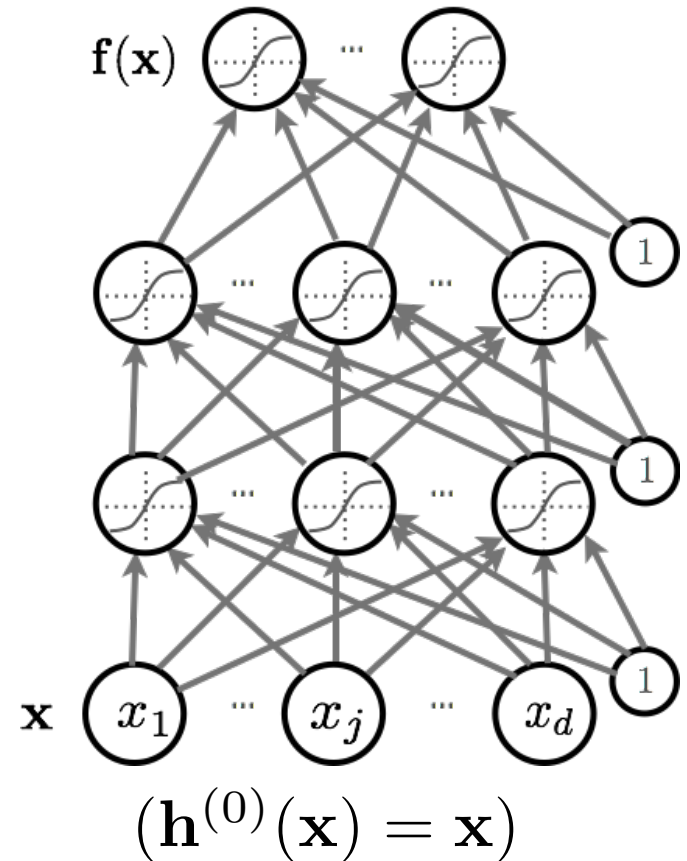
- hidden layer activation
from 1 to L :

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k = L + 1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

↖
Softmax activation
function



Gradient Computation

- Loss gradient at output

- Partial derivative:

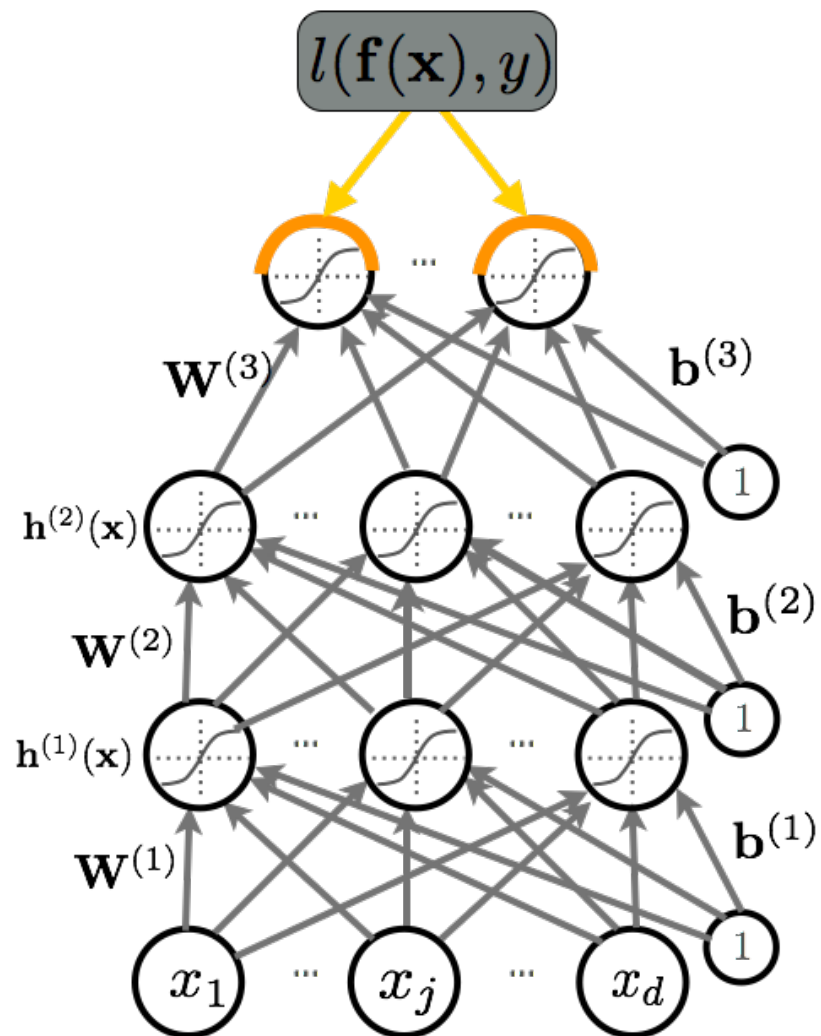
$$\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1_{(y=c)}}{f(\mathbf{x})_y}$$

- Gradient:

$$\begin{aligned} & \nabla_{\mathbf{f}(\mathbf{x})} - \log f(\mathbf{x})_y \\ &= \frac{-1}{f(\mathbf{x})_y} \begin{bmatrix} 1_{(y=0)} \\ \vdots \\ 1_{(y=C-1)} \end{bmatrix} \\ &= \frac{-\mathbf{e}(y)}{f(\mathbf{x})_y} \end{aligned}$$

Indicator function

Remember: $f(\mathbf{x})_c = p(y = c | \mathbf{x})$



Gradient Computation

- Loss gradient at output pre-activation

- Partial derivative:

$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y$$

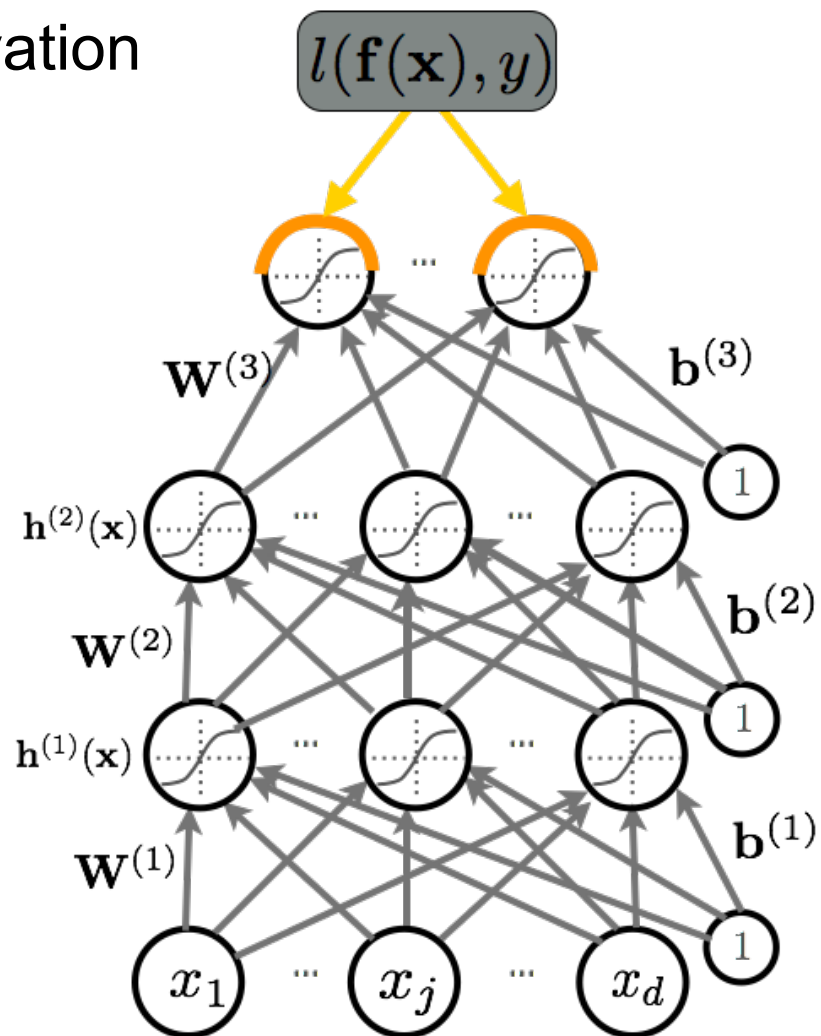
$$= - (1_{(y=c)} - f(\mathbf{x})_c)$$

- Gradient:

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= - (\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

Indicator function



Derivation

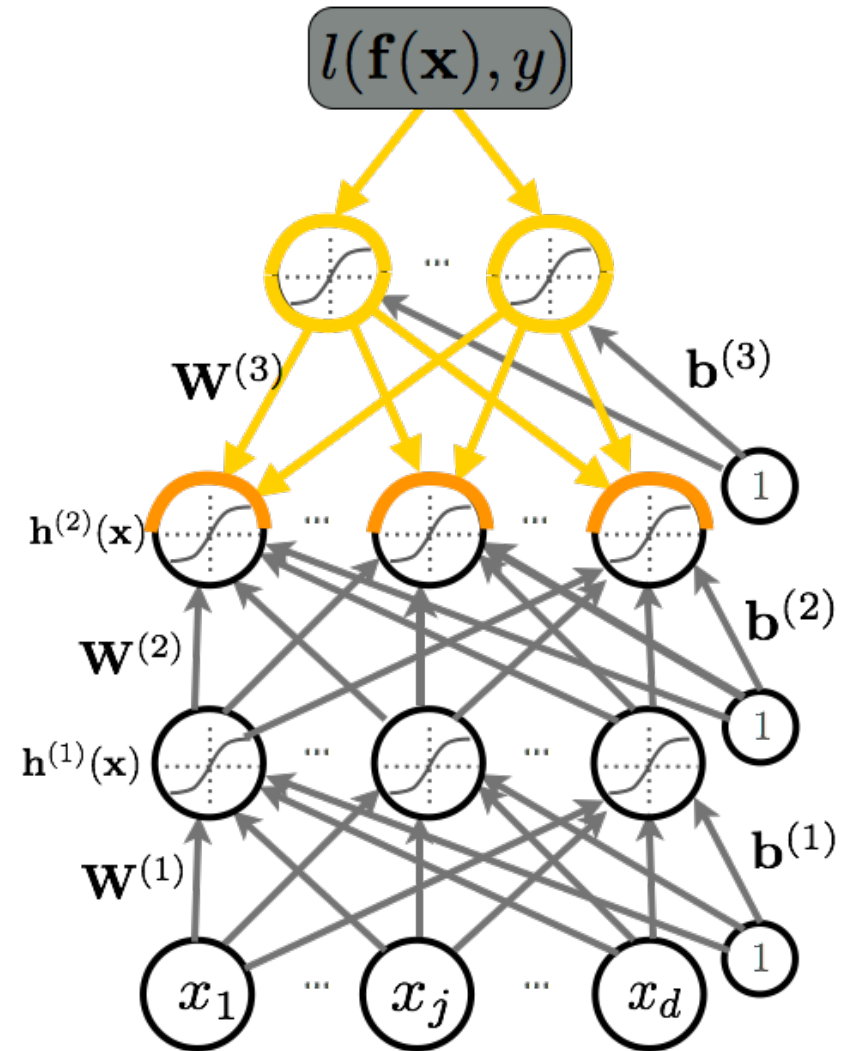
$$\frac{\partial \frac{g(x)}{h(x)}}{\partial x} = \frac{\partial g(x)}{\partial x} \frac{1}{h(x)} - \frac{g(x)}{h(x)^2} \frac{\partial h(x)}{\partial x}$$

$$\begin{aligned}
 & \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y \\
 = & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y \\
 = & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \\
 = & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \\
 = & \frac{-1}{f(\mathbf{x})_y} \left(\frac{\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y) \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)}{\left(\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)^2} \right) \\
 = & \frac{-1}{f(\mathbf{x})_y} \left(\frac{1_{(y=c)} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \frac{\exp(a^{(L+1)}(\mathbf{x})_c)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \right) \\
 = & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_c \right) \\
 = & \frac{-1}{f(\mathbf{x})_y} (1_{(y=c)} f(\mathbf{x})_y - f(\mathbf{x})_y f(\mathbf{x})_c) \\
 = & - (1_{(y=c)} - f(\mathbf{x})_c)
 \end{aligned}$$

Gradient Computation

- Loss gradient for **hidden layers**

– This is getting complicated!

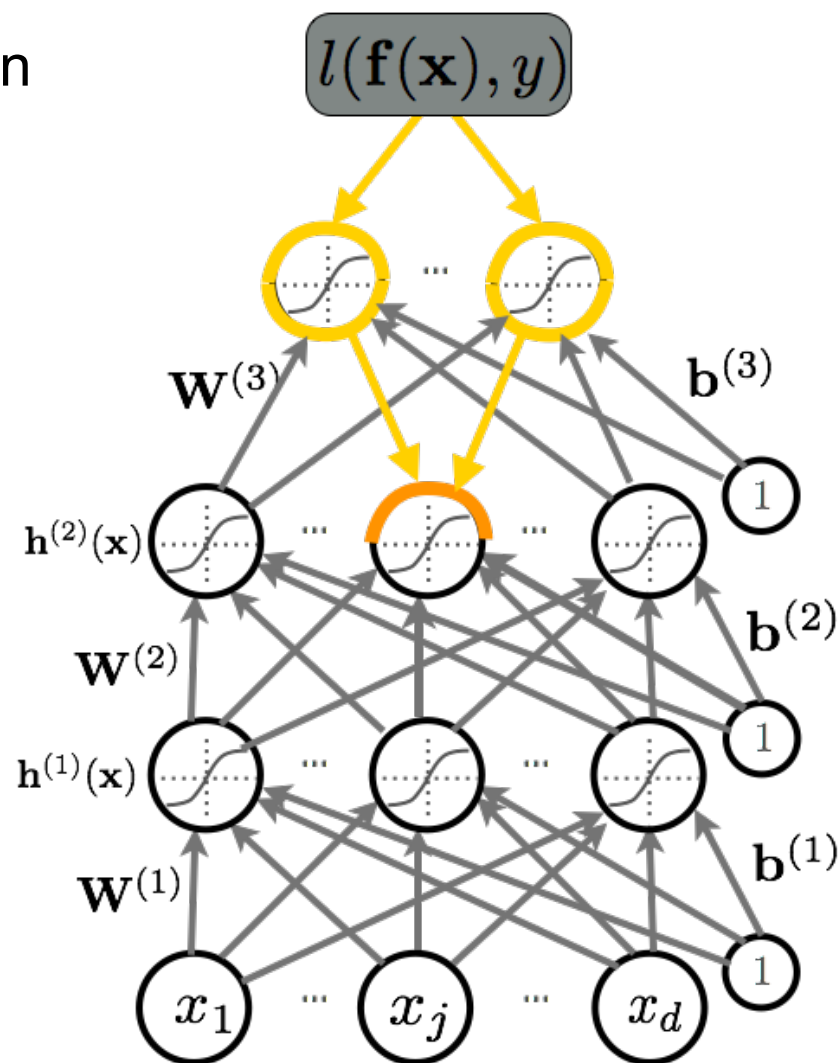


Gradient Computation

- **Chain Rule:** Assume that a function $p(a)$ can be written as a function of intermediate results $q_i(a)$, then:

$$\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$$

- We can invoke it by setting:
 - a be a hidden unit
 - $q_i(a)$ be a pre-activation in the layer above
 - $p(a)$ be the loss function



Gradient Computation

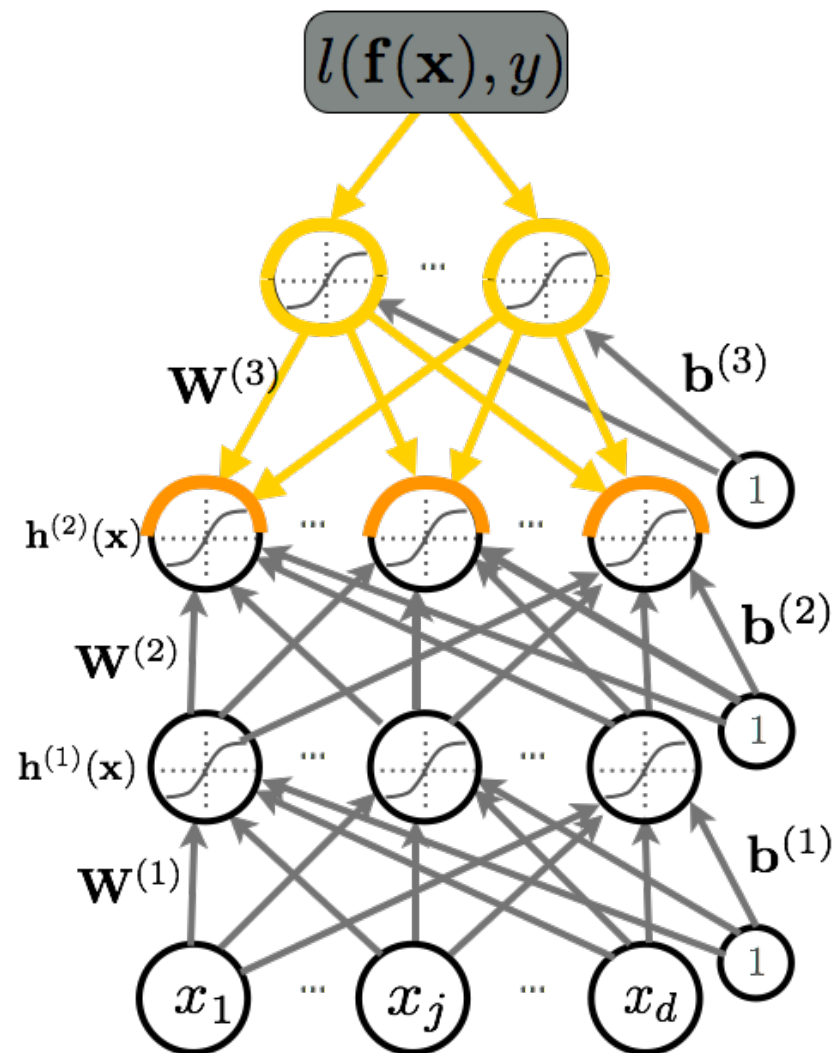
- Loss gradient at hidden layers

- Partial derivative:

$$\begin{aligned} & \frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y \\ &= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j} \\ &= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W_{i,j}^{(k+1)} \end{aligned}$$

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



Gradient Computation

- Loss gradient at hidden layers

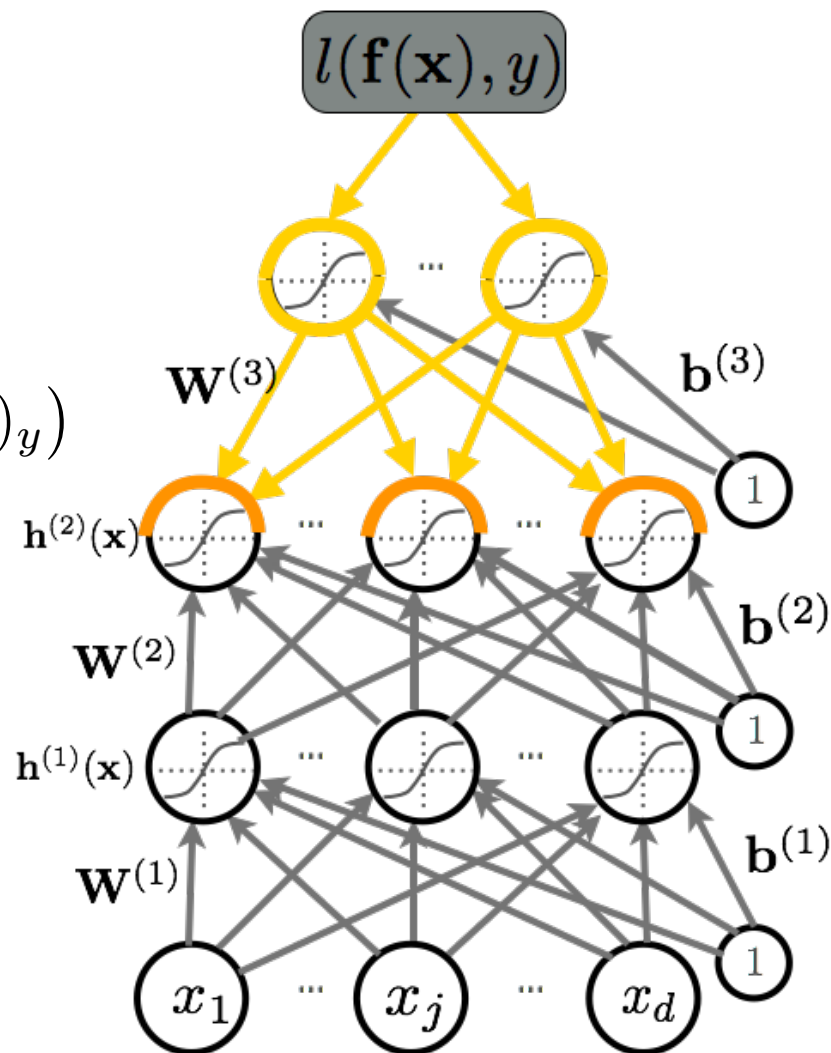
– Gradient

$$\begin{aligned} & \nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\ = & \mathbf{W}^{(k+1)\top} \underbrace{\left(\nabla_{\mathbf{a}^{(k+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \right)} \end{aligned}$$

We already know
how to compute
that

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



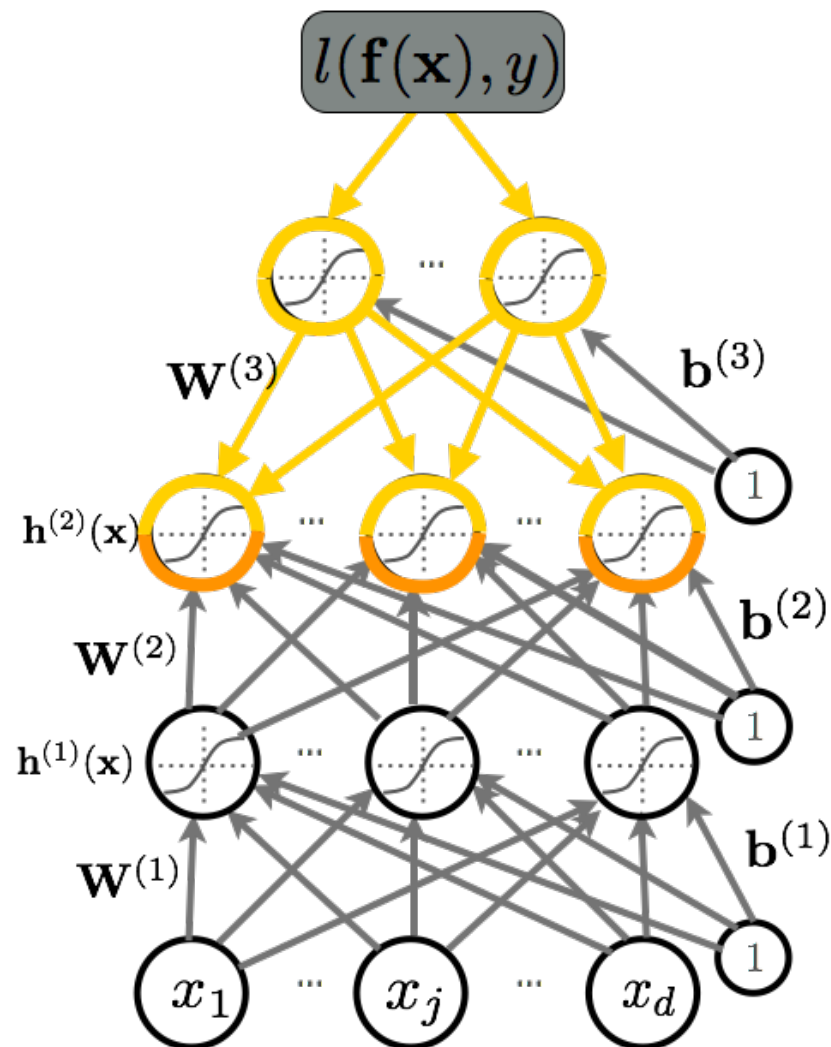
Gradient Computation

- Loss gradient at hidden layers (pre-activation)
 - Partial derivative:

$$\begin{aligned} & \frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y \\ = & \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j} \\ = & \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j) \end{aligned}$$

Remember:

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$



Gradient Computation

- Loss gradient at hidden layers (pre-activation)

– Gradient:

$$\begin{aligned} & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\ = & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \right)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\ = & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \right) \odot [\dots, g'(a^{(k)}(\mathbf{x})_j), \dots] \end{aligned}$$

Let's look at the gradients of activation functions.

Gradient of the
activation function

Remember:

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$

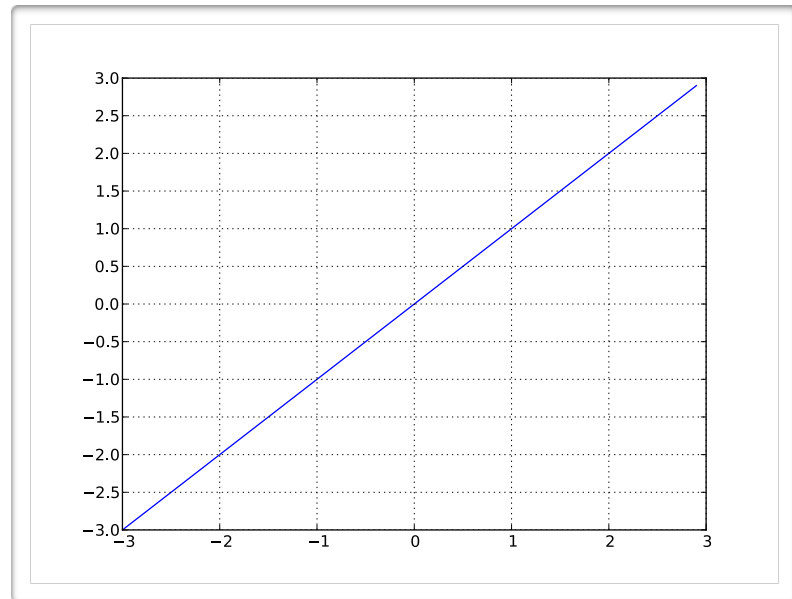
Linear Activation Function Gradient

- Linear activation function:

$$g(a) = a$$

- Partial derivative

$$g'(a) = 1$$



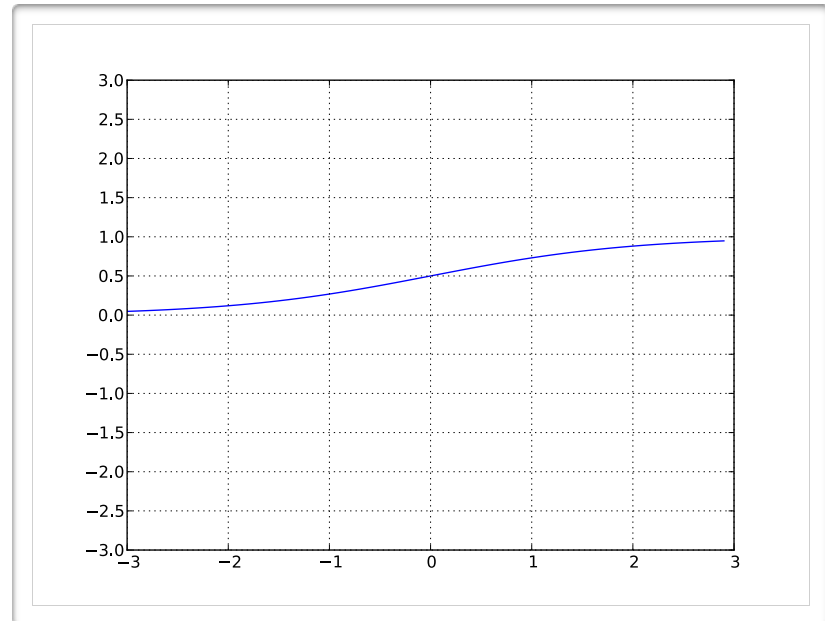
Sigmoid Activation Function Gradient

- Sigmoid activation function:

– Partial derivative

$$g'(a) = g(a)(1 - g(a))$$

$$g(a) = \text{sigm}(a) = \frac{1}{1 + \exp(-a)}$$



Tanh Activation Function Gradient

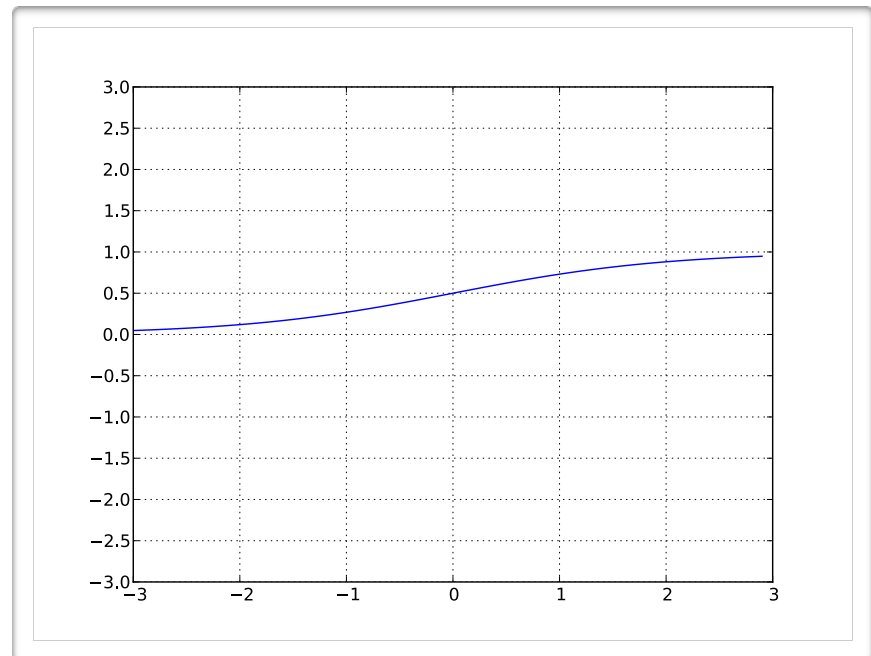
- Hyperbolic tangent (“tanh”) activation function:

$$g(a) = \tanh(a) =$$

– Partial derivative

$$= \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

$$g'(a) = 1 - g(a)^2$$



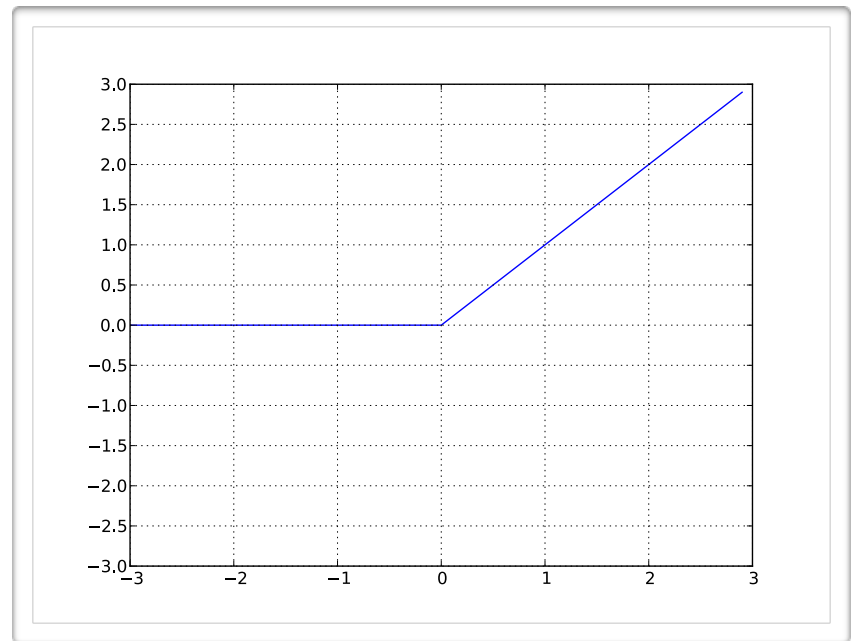
Tanh Activation Function Gradient

- Rectified linear (ReLU) activation function:

– Partial derivative

$$g'(a) = 1_{a>0}$$

$$g(a) = \text{reclin}(a) = \max(0, a)$$



Stochastic Gradient Descend

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For $t=1:T$

- for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:

- Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- A procedure to compute gradients: $\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- Regularizer and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

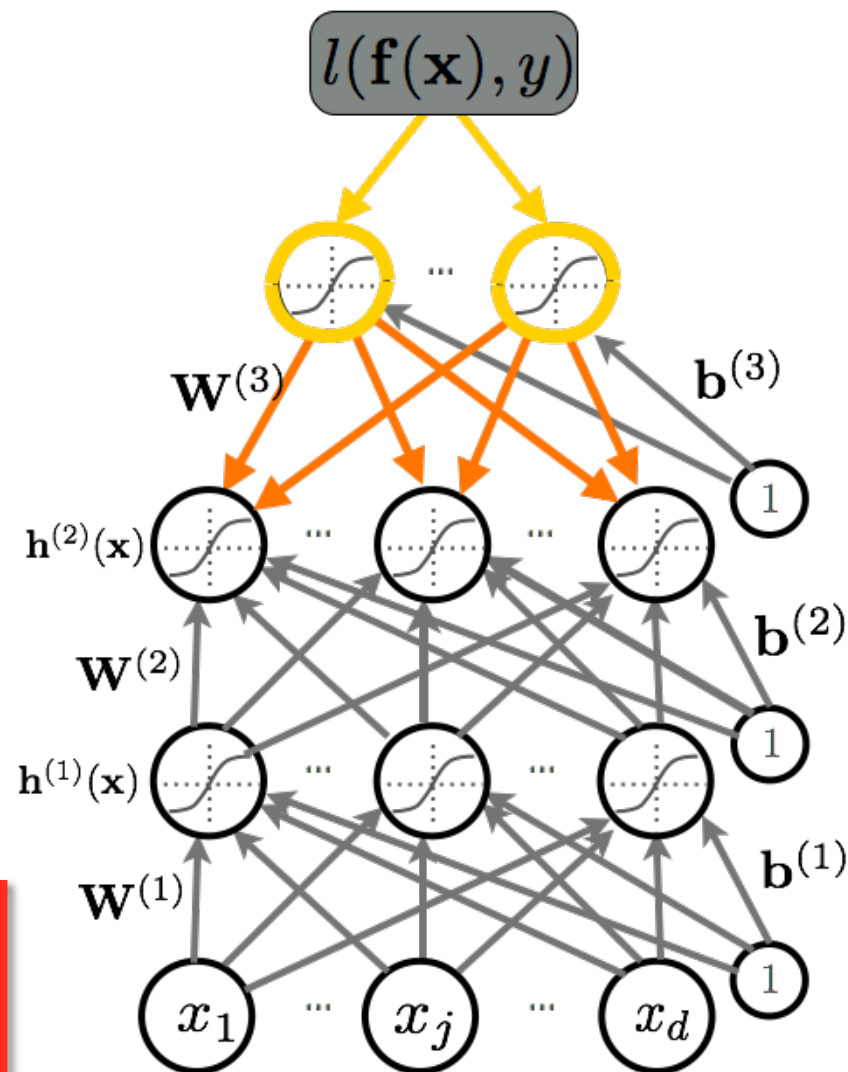
Gradient Computation

- Loss gradient of parameters
 - Partial derivative (weights):

$$\begin{aligned}
 & \frac{\partial}{\partial W_{i,j}^{(k)}} - \log f(\mathbf{x})_y \\
 = & \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial W_{i,j}^{(k)}} \\
 = & \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} h_j^{(k-1)}(\mathbf{x})
 \end{aligned}$$

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h_j^{(k-1)}(\mathbf{x})$$

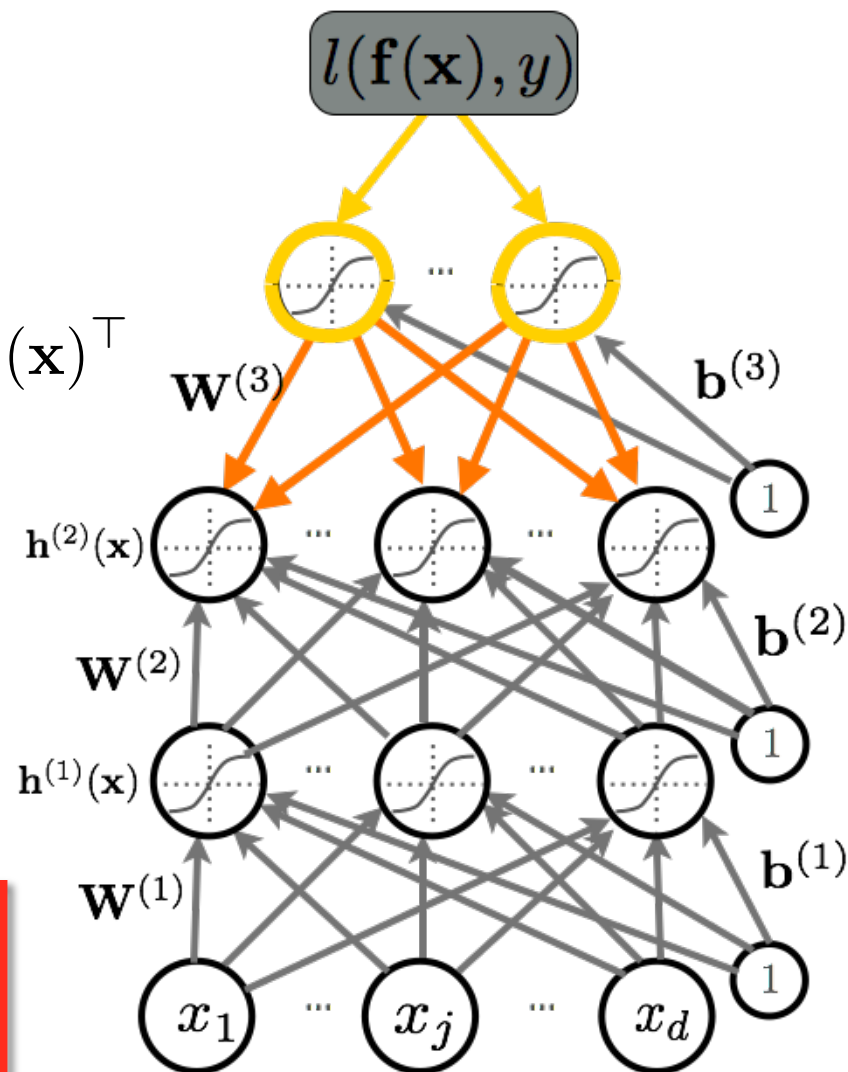


Gradient Computation

- Loss gradient of parameters

- Gradient (weights):

$$\begin{aligned} & \nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \\ = & \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \right) \mathbf{h}^{(k-1)}(\mathbf{x})^\top \end{aligned}$$



Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

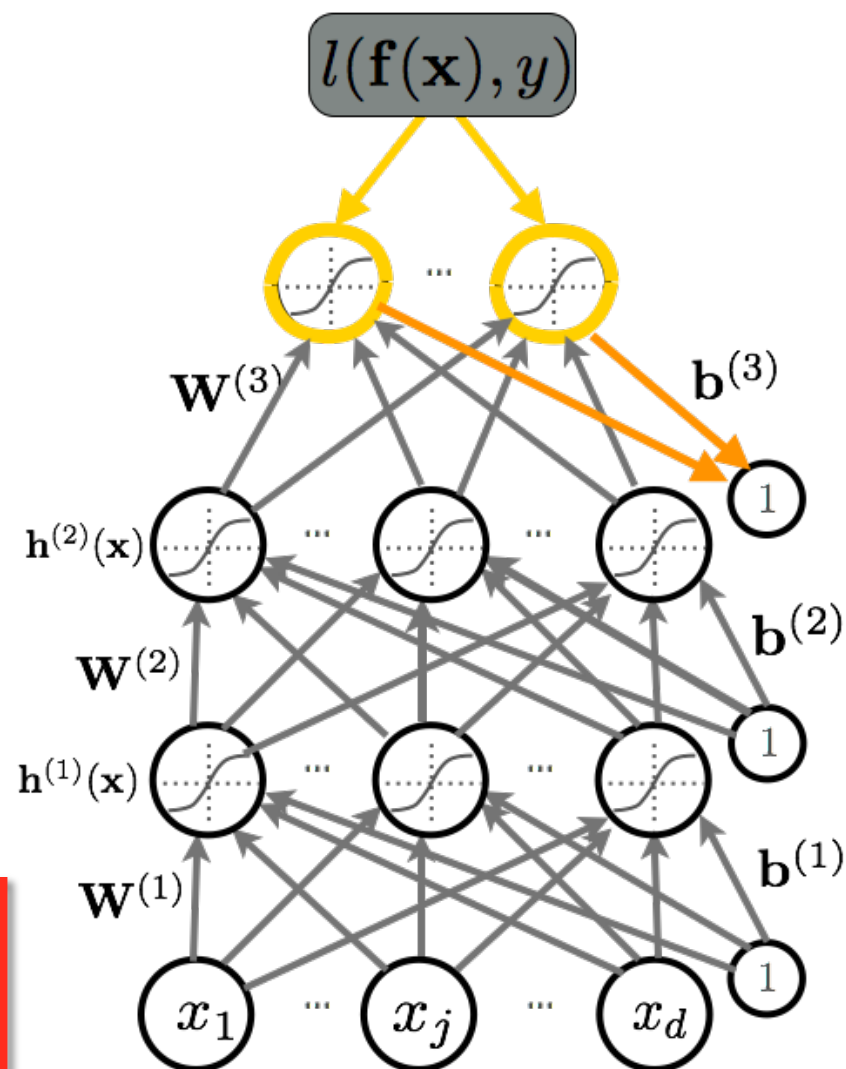
Gradient Computation

- Loss gradient of parameters
 - Partial derivative (biases):

$$\begin{aligned}
 & \frac{\partial}{\partial b_i^{(k)}} - \log f(\mathbf{x})_y \\
 = & \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial b_i^{(k)}} \\
 = & \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i}
 \end{aligned}$$

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



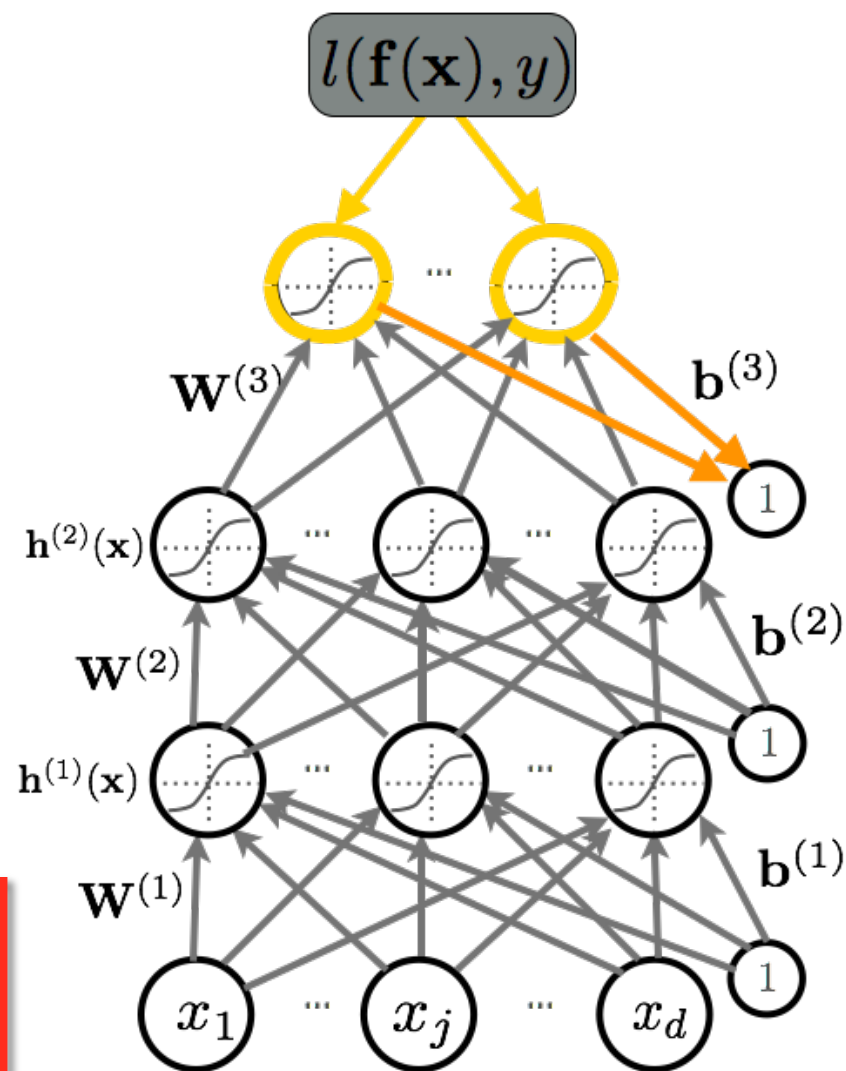
Gradient Computation

- Loss gradient of parameters
 - Gradient (biases):

$$\begin{aligned} & \nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \\ = & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \end{aligned}$$

Remember:

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



Backpropagation Algorithm

- Perform forward propagation
- Compute output gradient (before activation):

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Leftarrow -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

- For $k=L+1$ to 1

- Compute gradients w.r.t. the hidden layer parameters:

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \Leftarrow (\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y) \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \Leftarrow \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

- Compute gradients w.r.t. the hidden layer below:

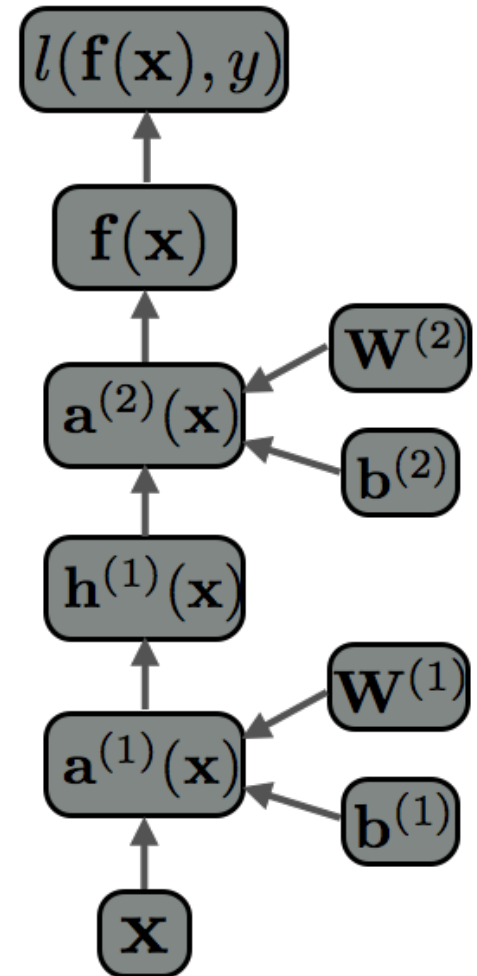
$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Leftarrow \mathbf{W}^{(k)\top} (\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)$$

- Compute gradients w.r.t. the hidden layer below (before activation):

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Leftarrow (\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y) \odot [\dots, g'(a^{(k-1)}(\mathbf{x})_j), \dots]$$

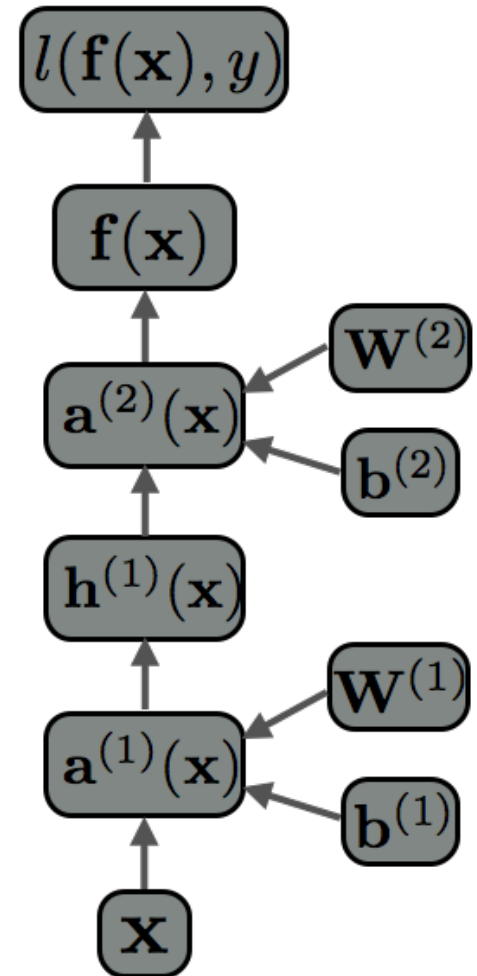
Computational Flow Graph

- Forward propagation can be represented as an acyclic flow graph
- Forward propagation can be implemented in a modular way:
 - Each box can be an object with an **fprop method**, that computes the value of the box given its children
 - Calling the fprop method of each box in the right order yields forward propagation



Computational Flow Graph

- Each object also has a **bprop method**
 - it computes the gradient of the loss with respect to each child box.
 - fprop depends on the fprop output of a box's children, while bprop depends on the bprop of a box's parents
- By calling bprop in the **reverse order**, we obtain backpropagation



Stochastic Gradient Descend

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For $t=1:T$

- for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

Training epoch
=
Iteration of all examples

- To train a neural net, we need:

- Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- A procedure to compute gradients: $\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}; \theta), y^{(t)})$

- Regularizer and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

Weight Decay

- L2 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)} \right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

- Gradient:

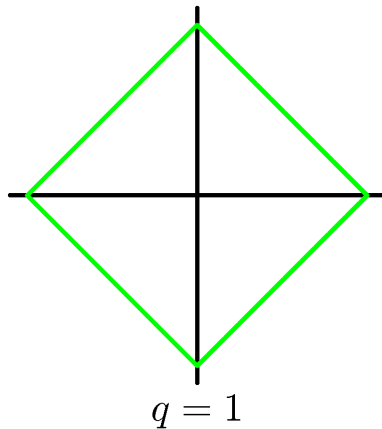
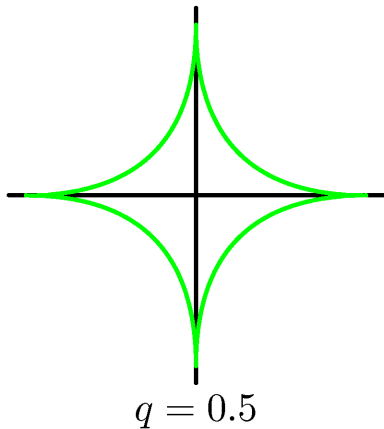
$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = 2\mathbf{W}^{(k)}$$

- Only applies to weights, not biases (weight decay)
- Can be interpreted as having a Gaussian prior over the weights, while performing MAP estimation.
- We will later look at Bayesian methods.

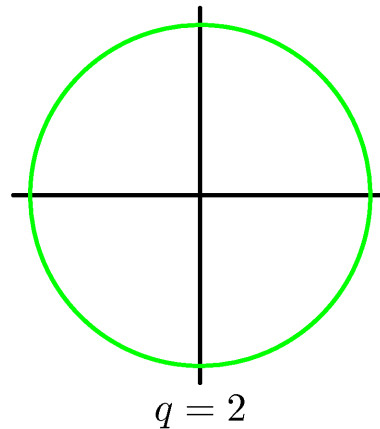
Other Regularizers

- Using a more general regularizer, we get:

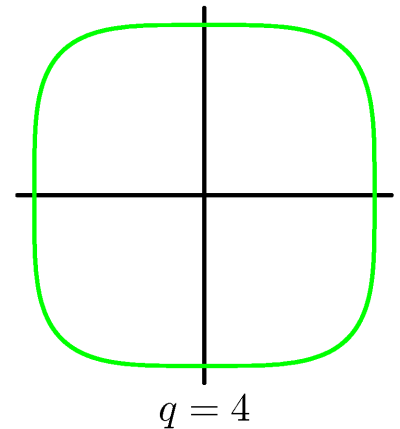
$$\frac{1}{2} \sum_{n=1}^N \{t_n - \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2} \sum_{j=1}^M |w_j|^q$$



Lasso



Quadratic



L1 Regularization

- L1 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$$

$$\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$$

- Only applies to weights, not biases (weight decay)
- Can be interpreted as having a Laplace prior over the weights, while performing MAP estimation.
- Unlike L2, L1 will push some weights to be exactly 0.

Bias-Variance Trade-off

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

Average predictions over all datasets differ from the optimal regression function.

Solutions for individual datasets vary around their averages -- how sensitive is the function to the particular choice of the dataset.

Intrinsic variability of the target values.

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x}) d\mathbf{x}$$

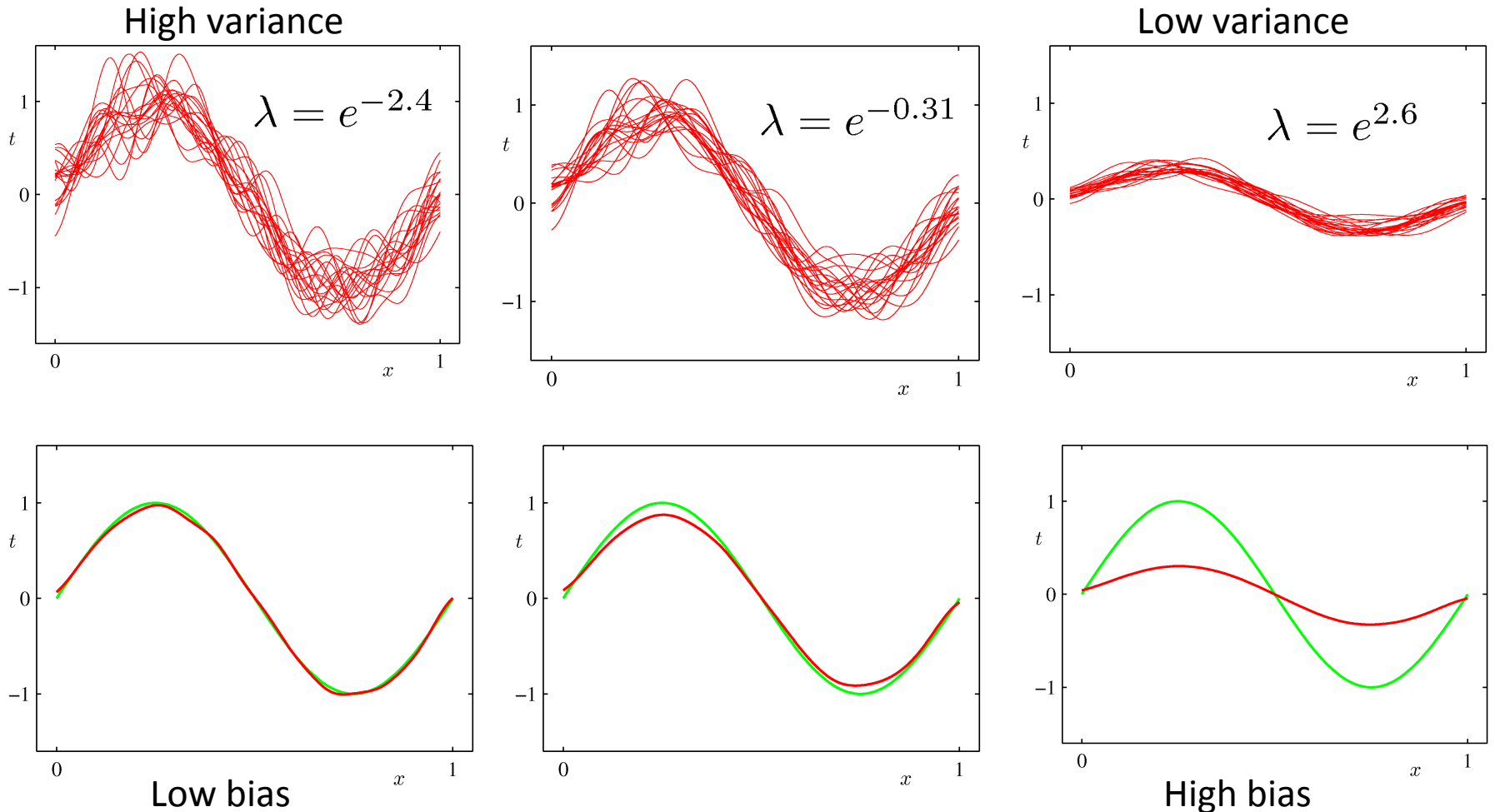
$$\text{variance} = \int \mathbb{E}_{\mathcal{D}} [\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}^2] p(\mathbf{x}) d\mathbf{x}$$

$$\text{noise} = \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t) d\mathbf{x} dt$$

- Trade-off between bias and variance: With very flexible models (high complexity) we have low bias and high variance; With relatively rigid models (low complexity) we have high bias and low variance.
- The model with the optimal predictive capabilities has to balance between bias and variance.

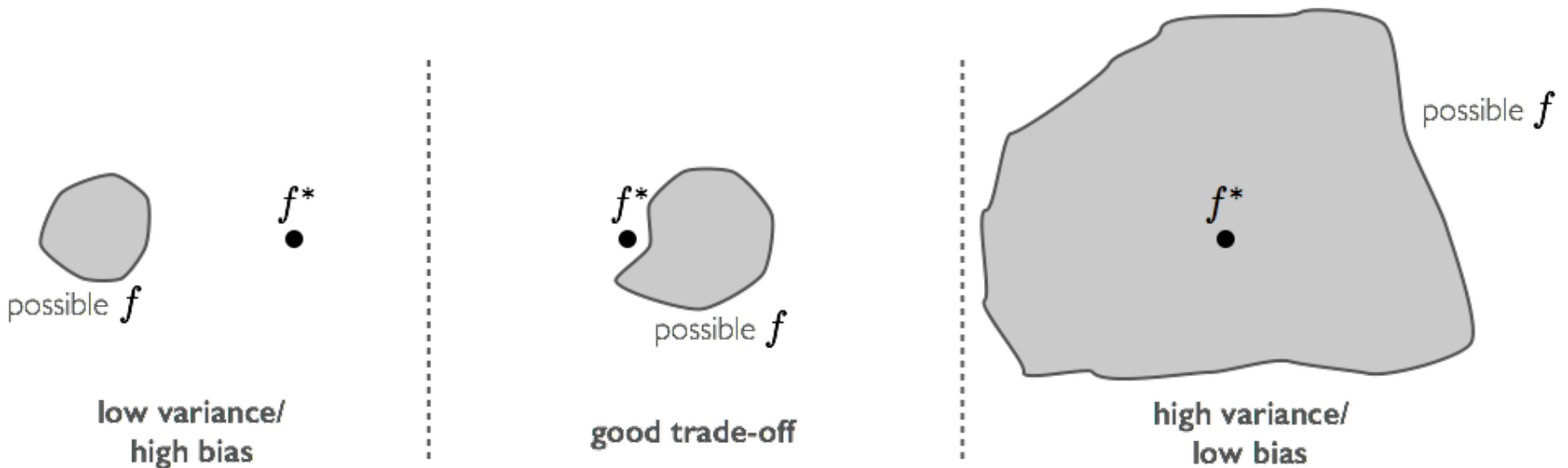
Bias-Variance Trade-off

- Consider the sinusoidal dataset. We generate 100 datasets, each containing $N=25$ points, drawn independently from $h(x) = \sin 2\pi x$.



Bias-Variance Trade-off

- **Generalization error** can be seen as the sum of the (squared) bias and the variance



Initialization

- Initialize biases to 0
- For weights
 - Can not initialize weights to 0 with tanh activation
 - All gradients would be zero (saddle point)
 - Can not initialize all weights to the same value
 - All hidden units in a layer will always behave the same
 - Need to break symmetry
 - Sample $\mathbf{W}_{i,j}^{(k)}$ from $U[-b, b]$, where

$$b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$$

Sample around 0 and
break symmetry



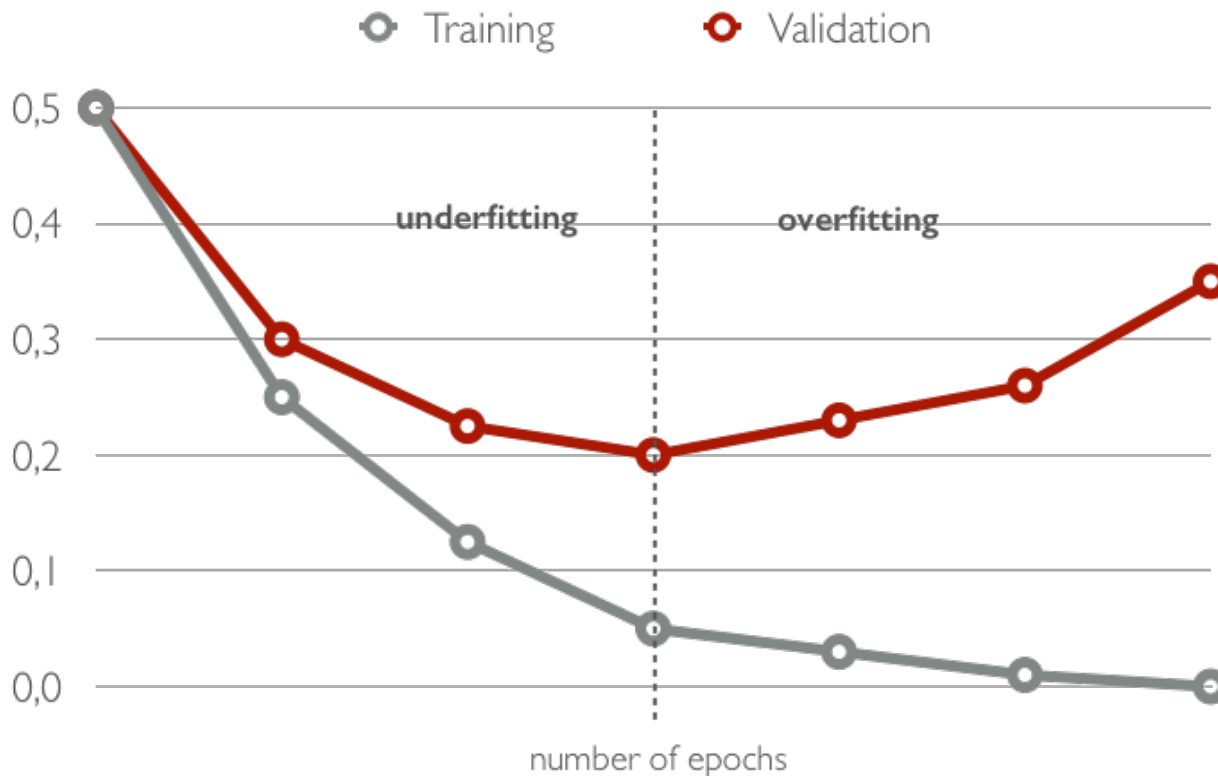
Size of $\mathbf{h}^{(k)}(\mathbf{x})$

Model Selection

- Training Protocol:
 - Train your model on the **Training Set** $\mathcal{D}^{\text{train}}$
 - For model selection, use **Validation Set** $\mathcal{D}^{\text{valid}}$
 - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.
 - Estimate generalization performance using the **Test Set** $\mathcal{D}^{\text{test}}$
- Remember: Generalization is the behavior of the model on **unseen examples**.

Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



Tricks of the Trade:

- Normalizing your (real-valued) data:
 - for each dimension x_i subtract its training set mean
 - divide each dimension x_i by its training set standard deviation
 - this can speed up training
- Decreasing the learning rate: As we get closer to the optimum, take smaller update steps:
 - i. start with large learning rate (e.g. 0.1)
 - ii. maintain until validation error stops improving
 - iii. divide learning rate by 2 and go back to (ii)

Mini-batch, Momentum

- Make updates based on a mini-batch of examples (instead of a single example):

- the gradient is the average regularized loss for that mini-batch
- can give a more accurate estimate of the gradient
- can leverage matrix/matrix operations, which are more efficient

- **Momentum**: Can use an exponential average of previous gradients:

$$\overline{\nabla}_{\theta}^{(t)} = \nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\theta}^{(t-1)}$$

- can get pass plateaus more quickly, by “gaining momentum”

Adapting Learning Rates

- Updates with adaptive learning rates (“one learning rate per parameter”)

- **Adagrad**: learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\gamma^{(t)} = \gamma^{(t-1)} + \left(\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) \right)^2 \quad \bar{\nabla}_{\theta}^{(t)} = \frac{\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

- **RMSProp**: instead of cumulative sum, use exponential moving average

$$\gamma^{(t)} = \beta \gamma^{(t-1)} + (1 - \beta) \left(\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) \right)^2$$

- **Adam**: essentially combines RMSProp with momentum

$$\bar{\nabla}_{\theta}^{(t)} = \frac{\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

Gradient Checking

- To debug your implementation of fprop/bprop, you can compare with a finite-difference approximation of the gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

- $f(x)$ would be the loss
- x would be a parameter
- $f(x + \epsilon)$ would be the loss if you add ϵ to the parameter
- $f(x - \epsilon)$ would be the loss if you subtract ϵ to the parameter

Debugging on Small Dataset

- Next, make sure your model can overfit on a smaller dataset (~ 500-1000 examples)
- If not, investigate the following situations:
 - Are some of the units **saturated**, even before the first update?
 - scale down the initialization of your parameters for these units
 - properly normalize the inputs
 - Is the training error bouncing up and down?
 - decrease the learning rate
- This does not mean that you have computed gradients correctly:
 - You could still overfit with some of the gradients being wrong